The Chord protocol and Dynamic Membership

INF-3200: Distributed Systems Fundamentals

Kevin M. Bergan
kbe153@uit.no

I. Introduction

This report details the implementation of a dynamic distributed key-value store network, building upon the system developed in Assignment 1. The network is structured as an overlay based on the Chord protocol [1], where nodes are ordered by their hashed identifiers and maintain pointers to their successor nodes. The focus of this assignment is to extend the functionality by enabling nodes to join and leave the network dynamically, while still providing support for key-value storage operations using HTTP-based APIs. The API extensions include mechanisms for nodes to join, leave, and simulate crash scenarios, while also maintaining the ability to execute PUT and GET requests for key-value pairs.

The system i designed to operate across the ifi cluster, allowing each node to communicate and adjust to changes in the network topology. A new set of HTTP API calls was introduced to facilitate these dynamic behaviors, including */join*, */leave*, */sim-crash*, and */sim-recover*. These APIs enable nodes to participate in the network, exit cleanly, or handle failure situations, with the rest of the network adjusting accordingly. Additionally, nodes provide metadata about their state and neighbors through a */node-info endpoint*. This report outlines the design, implementation challenges, and testing of these dynamic features.

II. Technical Background

This assignment requires a solid understanding of distributed systems, particularly in terms of implementing dynamic behavior in a distributed key-value store. Nodes must be able to join and leave the network without being initialized with knowledge of other nodes. Each node begins in isolation and must rely on API calls to integrate into the existing ring topology. The key technical challenge is ensuring that nodes can dynamically update their successors, finger tables, and routing mechanisms when the network changes, while maintaining the correct key-value mapping for PUT and GET operations. For my system, i have not integrated data transfer between nodes i.e. the key-value pairs remains on the node that they were first stored on. The system should remain functional during stable periods, but it's acceptable for instability to arise after nodes leave or crash.

III. Design & Implementation

The system's design begins by initializing each node as a self-contained entity with no initial connections, ensuring that nodes start in isolation. When a node wishes to join the network, it sends a POST request to a target node (e.g., "node1/join?nprime=node2"). The target node processes the request, determines the appropriate position for the new node, and updates the successor and predecessor pointers for the joining node and its immediate neighbors. This approach allows the network to dynamically adjust as nodes join, relying only on local information, thus preserving network integrity and avoiding the need for global knowledge.

The use of consistent hashing ensures smooth integration, allowing key-value mappings to remain stable as the network grows. Scalability is supported as nodes can be added without significant overhead. In the Chord protocol, each node maintains a finger table with references to other nodes at specific intervals, typically calculated as $n+2^k$, which enables efficient key lookups. This structure allows a node in an N-node network to find a successor in $O(\log N)$ time.

To maintain network stability, nodes periodically update their finger tables through the *periodicUpdateFingerTable()*function, which triggers three key procedures: stabilize(), *checkPredecessor()*, and *updateFingerTable()*. The stabilize() function verifies the correctness of the node's successor, while *updateFingerTable()* recalculates the correct finger table entries based on Chord's consistent hashing. Lastly, *checkPredecessor()* ensures that failed predecessors are handled appropriately. Together, these procedures maintain the system's robustness as nodes join or leave the network.

IV. Experiments

Figure 1 illustrates the average time taken to join and leave nodes in a distributed network, with a keyspace size $m = 10$, over 20 rounds. For join operations, the time appears relatively stable as the network size increases, starting around 1.75 milliseconds for a network of 2 nodes and reaching approximately 2.0 milliseconds for larger network sizes. However, the error bars show that the variability in join times grows as the network expands, especially in the cases with 32 and 64 nodes. This suggests that while the average join time does not significantly increase, the performance becomes less predictable as the number of nodes in the network grows.

I have created two test scripts that checks whether the distributed system can handle joins from 32 single nodes all the way to 32 connected nodes. Once reached the full network, the script make each node leave one by one all the way to the original set of 32 single nodes. This pattern continues 20 more times resulting in good data to analyze. This is illustrated in figure 1.

There are some artifacts that might arise when testing my implementation of dynamic joining and leaving a distributed network. One error i get is that whenever a node sends a GET request to another node, that request takes such a long time that it times out, resulting in an error. This only seems to happens on network sizes 32 and above.

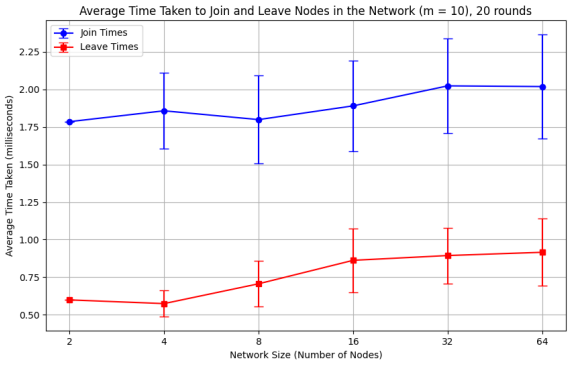

Fig. 1. Vector Graph where keyspace m = 10 and test rounds = 20

V. Discussion

There are notable concerns that could impact both reliability and performance in the system. One issue is the reliance on synchronous HTTP requests to forward GET and PUT operations to successor nodes. While this design simplifies request propagation, it introduces latency and could lead to bottlenecks, particularly in large-scale systems with high traffic, which i think my network is suffering from. In cases of network failure or significant delays, the system could experience timeouts or become unresponsive, as demonstrated by the strict time limits (10 seconds) set for client connections. Moreover, the decision to treat a node's failure by returning HTTP 503 status codes may not be sufficient for ensuring fault tolerance. A more robust failure detection and recovery mechanism, such as utilizing a quorum-based protocol [2] or distributed consensus for handling node crashes [3], could improve the system's resilience.

Another area that presents potential challenges is the wrap-around handling logic in the ring structure. While my code accounts for cases where the predecessor node has a higher identifier than the current node, there is room for optimization in how key ranges are handled. The current approach may lead to inefficiencies when a key falls between predecessor and current node boundaries, especially in cases involving frequent node joins or departures. This could cause unnecessary forwarding of requests or incorrect data placement if the node relationships are not properly synchronized. A more sophisticated mechanism for key range partitioning, such as leveraging finger tables more effectively, might address these issues and improve overall system efficiency.

VI. Conclusion

In conclusion, the design and implementation of the distributed key-value store system, structured as an overlay network based on the Chord protocol, successfully supports the dynamic joining and leaving of nodes. Each node initializes as an isolated entity, allowing for independent operation within the network. By utilizing additional HTTP API calls, nodes can seamlessly join the rings of other nodes, ensuring that the network adapts fluidly to changes in membership. The consistent hashing mechanism guarantees that PUT operations correctly associate data with the appropriate nodes, and the retrieval of values via GET operations remains reliable as long as the network remains stable. Although temporary unavailability of data may occur if a node leaves, the system is designed to maintain the integrity of finger tables through periodic updates, ensuring efficient key lookups with a logarithmic time complexity of $O(logN)$. Overall, this implementation demonstrates a robust and flexible architecture capable of handling dynamic network changes while fulfilling the assignment's requirements.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM transactions on networking*, vol. 11, no. 1, pp. 17–32, 2003.

[2] D. Skeen, "A quorum-based commit protocol," Cornell University, Tech. Rep., 1982.

[3] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Distributed Computing: 12th International Symposium, DISC'98 Andros, Greece, September 24–26, 1998 Proceedings 12*. Springer, 1998, pp. 231–245.